Online Companion Caching

Amos Fiat[1], Manor Mendel[2], and Steven S. Seiden[3]*

[1] School of Computer Science, Tel-Aviv University
fiat@tau.ac.il
[2] School of Computer Science, The Hebrew University
mendelma@cs.huji.ac.il
[3] Department of Computer Science, Louisiana State University, Baton Rouge

Steve Seiden died in a tragic accident on June 11, 2002. The other authors would like to dedicate this paper to his memory.

Abstract. This paper is concerned with online caching algorithms for the (n, k)-companion cache, defined by Brehob *et. al.* [3]. In this model the cache is composed of two components: a k-way set-associative cache and a companion fully-associative cache of size n. We show that the deterministic competitive ratio for this problem is $(n+1)(k+1)-1$, and the randomized competitive ratio is $O(\log n \log k)$ and $\Omega(\log n + \log k)$.

1 Introduction

A popular cache architecture in modern computer systems is the *set-associative cache*. In a k-way set-associative cache, a cache of size s is divided into $m = s/k$ disjoint sets, each of size k. Addresses in main memory are likewise assigned one of m types, and the i'th associative cache can only store memory cells whose address is type i. Special cases includes *direct-mapped caches*, which are 1-way set associative caches, and fully-associative caches, which are s-way set associative caches.

In order to overcome "hot-spots", where the same set associative cache is being constantly accessed, computer architects have designed hybrid cache architectures. Typically such a cache has two or more components. A given item can be placed in any of the components of the cache. Brehob *et. al.* [3] considered the (n, k) companion cache, which consists of two components: A k-way set associative called the *main cache*, and a fully-associative cache of size n, called the *companion cache* (the names stem from the fact that typically $mk \gg n$). As argued by Brehob *et. al.* [3], many of the L1-cache designs suggested in recent years use companion caches as the underlying architecture. Several variations on the basic companion cache structure are possible. These include reorganization/no-reorganization and bypassing/no-bypassing. Reorganization is the ability to move an item from one cache component to another, whereas bypassing is the ability to avoid storing an accessed item in the cache. A schematic view of the companion cache is presented in Fig. 1.

* This research was partially supported by the Louisiana Board of Regents Research Competitiveness Subprogram and by AFOSR grant No. F49620-01-1-0264.

R. Möhring and R. Raman (Eds.): ESA 2002, LNCS 2461, pp. 499–511, 2002.
© Springer-Verlag Berlin Heidelberg 2002

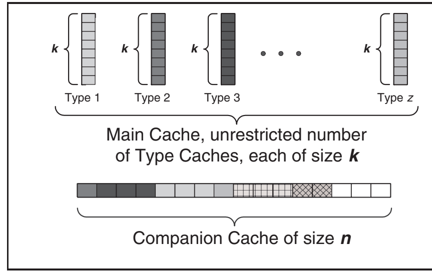

Fig. 1. A schematic description of a companion cache

Since maintenance of the cache must be done online, and this makes it impossible to service requests optimally, we use *competitive analysis*. The usual assumption is that any referenced item is brought into the cache before it is accessed. Since items in the cache are accessed much more quickly than those outside, we associate costs with servicing items as follows: If the referenced item is already in the cache then we say that the reference is a *hit* and the cost is zero. Otherwise, we have a *fault* or *miss* which costs one. Roughly speaking, an online caching algorithm is called r-competitive if for any request sequence the number of faults is at most r times the number of faults of the optimal offline algorithm, allowing a constant additive term.

Previous Results: Maintenance of a fully associative cache of size k is the well known *paging problem* [2]. Sleator and Tarjan [8] proved that natural algorithms such as Least Recently Used are k-competitive, and that this is optimal for deterministic online algorithms. Fiat *et. al.* [4], improved by McGeoch and Sleator [7] and Achlioptas *et. al.* [1], show a tight $\approx \ln k$ competitive randomized algorithm. k-way set associative caches can be viewed as a collection of independent fully associative caches, each of size k, and therefore they are uninteresting algorithmically. Brehob *et. al.* [3] study deterministic online algorithms for $(n, 1)$-companion caches. They investigate the four previously mentioned variants, i.e., bypassing/no-bypassing and reorganization/no-reorganization.

Our Results: This paper studies deterministic and randomized caching algorithms for a (n, k)-companion cache. We consider the version where reorganization is allowed but bypassing is not.We show that the deterministic competitive ratio is exactly $(n + 1)(k + 1) - 1$. For randomized algorithms, we present an upper bound of $O(\log n \log k)$ on the competitive ratio, and a lower bound of $\Omega(\log n + \log k)$. For the special case of $k = 1$ that was studied in [3], our bounds on the randomized competitive ratio are tight up to a constant factor. The results of [3] and those of this paper are summarized and compared in Table 1.

We note that any algorithm for the reorganization model can be implemented (in online fashion) in the no-reorganization model while incurring a cost at most three times larger, and any algorithm for the bypassing model can be implemented (in online fashion) in the no-bypassing model while incurring a cost

Table 1. Summary of the results in [3] and in this paper. The size of the companion cache is n

Previous Results [3] (only for main cache of size $k = 1$):

Bypass	Reorg'	det'/rand'	Upper Bound	Lower Bound
–	–	det	$2n + 2$	$n + 1$
$\surd$	–	det	$2n + 3$	$2n + 2$
$\surd$	$\surd$	det	$2n + 3$	

New Results (main cache of arbitrary size k):

Bypass	Reorg'	det'/rand'	Upper Bound	Lower Bound
–	$\surd$	det	$(n + 1)(k + 1) - 1$	$(n + 1)(k + 1) - 1$
$-/\surd$	$-/\surd$	det	$O(nk)$	$\Omega(nk)$
$-/\surd$	$-/\surd$	rand	$O(\log k \log n)$	$\Omega(\log k + \log n)$

at most two times larger. Thus, the competitive ratio (both randomized and deterministic) differs by at most constant factor between the different models.

The techniques we use generalize *phase partitioning* and *marking algorithms* [6, 4].

2 The Problem

In the (n, k)-companion caching problem, there is a slow *main memory* and a fast *cache*. The items in main memory are partitioned into m *types*, the set of types is T ($|T| = m$). The cache consists of a two separate components:

- The Main Cache: Consisting of a cache of size k for each type. *I.e.*, every type t, $1 \leq t \leq m$, has its own cache of size k which can hold only items of type t.
- The Companion Cache: A cache of size n which can hold items of any type.

We refer to these components collectively simply as *the cache*. If an item is stored somewhere in the cache, we say it is *cached*. Our basic assumptions are that there are at least $k+1$ items of every type and that the number of types, m, is greater than the size of the companion cache, n.

A caching algorithm is faced with a sequence of requests for items. When an item is requested it must be cached (*i.e.*, bypassing is not allowed). If the item is not cached, a fault occurs. The goal is to minimize the number of faults. A caching algorithm can swap items of the same type between the main and companion caches without incurring any additional cost (*i.e.*, reorganization is allowed).

We use the competitive ratio to measure the performance of online algorithms. Formally, given an item request sequence σ, the cost of an online algorithm A on σ, denoted by $\text{cost}_A(\sigma)$, is the number of faults incurred by A. An

algorithm is called r-competitive if there exists a constant c, such that for any request sequence σ, $E[\text{cost}_A(\sigma)] \leq r \cdot \text{cost}_{\mathsf{Opt}}(\sigma) + c$.

To simplify the analysis later, we mention the following fact (attributed to folklore):

Proposition 1. *We may assume that* Opt *is lazy, i.e.,* Opt *evicts an item only when a requested item is not cached.*

Straightforward lower bounds follow from the classical paging problem.

Theorem 1. *The deterministic competitive ratio for the (n, k)-companion caching problem is at least $(n + 1)(k + 1) - 1$. The randomized competitive ratio is at least $H_{(k+1)(n+1)-1} = \Omega(\log n + \log k)$.*

Proof. Consider the situation where there are $(n+1)(k+1)$ items of $n+1$ types, $k + 1$ items of each type. In this case, a caching algorithm has $(n + 1)k + n = (n + 1)(k + 1) - 1$ cache slots available. If we compare this situation to the regular paging problem where the virtual memory consists of $(n+1)(k+1)$ pages (items) and the cache is of size $(n + 1)(k + 1) - 1$, we find the two problems are exactly the same. A companion caching algorithm induces a paging algorithm, and the opposite is also true. Hence a lower bound on the competitive ratio for paging implies the same lower bound for companion caching. We conclude there are lower bounds of $(n + 1)(k + 1) - 1$ on the deterministic competitive ratio and $H_{(n+1)(k+1)-1} = \Omega(\log n + \log k)$ on the randomized competitive ratio for companion caching. □

3 Phase Partitioning of Request Sequences

In [6, 4] the request sequence for the paging problem is partitioned into *phases* as follows: A phase begins either at the beginning of the sequence or immediately after the end of the previous phase. A phase ends either at the end of the sequence or immediately before the request for the $(k + 1)$st distinct page in the phase. Similarly, we partition the request sequence for the companion caching problem into phases. However, the more complex nature of our problem implies more complex partition rules.

Let $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_{|\sigma|}$ denote the request sequence. The indices of the sequence are partitioned into a sequence of disjoint consecutive subsequences $D_1, D_2, \ldots, D_f$, whose concatenation gives $\{1, \ldots, |\sigma|\}$. The indices are also partitioned into a sequence of disjoint (ascending) subsequences $P_1, P_2, \ldots, P_f$.

In Fig. 2 we describe how to generate the sequences D_i and P_i. D_i is a consecutive sequence of indices of requests *issued during* phase i. P_i is a (possibly non-consecutive, ascending) sequence of indices of requests *associated* with phase i.

Given a set of indices A we denote by $\mathbb{I}(A) = \{\sigma_\ell | \ell \in A\}$ the set of items requested in A, and by $\mathbb{T}(A)$ the set of types of items in $\mathbb{I}(A)$. Table 2 shows an example of phase partitioning.

In [6] it is shown that any paging algorithm faults at least once in each complete phase. Here we show a similar claim for companion caching.

P_i: The indices of the requests *associated with* phase i.
D_i: The indices of the requests issued *during* phase i.
$N(t)$: The indices of requests of type t that have not yet been associated with a phase.
$M(t)$ $= \{\sigma_\ell | \ell \in N(t)\}$

For every type $t \in T$: $M(t) \leftarrow \emptyset$, $N(t) \leftarrow \emptyset$
$P_1 \leftarrow \emptyset$, $D_1 \leftarrow \emptyset$
$i \leftarrow 1$
For $\ell \leftarrow 1, 2, \ldots$ *Loop on the requests*
  Let σ_ℓ be the current request and t_0 be its type.
  Let $m_t \leftarrow \begin{cases} \max\{0, |M(t)| - k\} & t \neq t_0 \\ \max\{0, |M(t_0) \cup \{\sigma_\ell\}| - k\} & t = t_0 \end{cases}$
  If $\sum_{t \in T} m_t > n$ then *End of Phase Processing:*
    For every type $t \in T$ such that $m_t > 0$ do
      $P_i \leftarrow P_i \cup N(t)$
      $M(t) \leftarrow \emptyset$, $N(t) \leftarrow \emptyset$
    $i \leftarrow i + 1$
    $P_i \leftarrow \emptyset$, $D_i \leftarrow \emptyset$
  $D_i \leftarrow D_i \cup \{\ell\}$
  $N(t_0) \leftarrow N(t_0) \cup \{\ell\}$
  $M(t_0) \leftarrow M(t_0) \cup \{\sigma_\ell\}$

Fig. 2. Phase partition rules described as an algorithm

Proposition 2. *For any (online or offline) caching algorithm, it is possible to associate with each phase (except maybe the last one) a distinct fault.*

Proof. Consider the request indices in P_i together with the index j that ends the phase (*i.e.*, $j = \min D_{i+1}$). One of the items in $\mathbb{I}(P_i)$ must be evicted after being requested and before σ_j is served. This is simply because the cache can not hold all these items simultaneously. We associate this eviction with the phase.

We must show that we have not associated the same eviction to two distinct phases. Let i_1 and i_2 be two distinct phases, $i_1 < i_2$. If the evictions associated with i_1 and i_2 are of different items then they are obviously distinct. Otherwise, the evictions associated with i_1 and i_2 are of the same type t, and $t \in \mathbb{T}(P_{i_1}) \cap \mathbb{T}(P_{i_2})$, which means that all indices $\ell \in P_{i_2}$, where σ_ℓ is of type t, must have $\ell > \max D_{i_1}$. Thus, an eviction associated with phase i_2 cannot be associated with phase i_1. □

To help clarify our argument in the proof of Proposition 2, consider the third phase in Table 2. Here $\mathbb{I}(P_3) = \{b_4, b_1, b_2, b_5, d_1, d_2\}$, and the phase ends because of the request to d_3. It is not possible that all these items reside in the cache simultaneously and thus at least one of the items in $\mathbb{I}(P_3)$ must be evicted before or on the request for item d_3. The item evicted can be either some b_i, $i = 1, 2, 4, 5$, or some d_i, $i = 1, 2$. If, for example, the item evicted is some b_i, then this eviction

Table 2. An example for an (n,k)-companion caching problem where $n=3$ and $k=2$. The types are denoted by the letters a,b,c,d. The ith item of type $\beta \in \{a,b,c,d\}$ is denoted by β_i

Req. seq.	$a_1 b_1 d_1 c_1 a_2 a_3 b_2 a_4 b_3 c_2$	$b_4 a_5 c_3 d_2 b_1 c_4 a_3 a_2$	$a_1 a_3 b_2 b_4 b_5$	$d_3 \ldots$
Phase	$i=1$	$i=2$	$i=3$	
D_i	$\{1,\ldots,10\}$	$\{11,\ldots,18\}$	$\{19,\ldots,23\}$	
P_i	$\{1,2,5,6,7,8,9\}$	$\left\{\begin{matrix}4,10,12,13,\\16,17,18\end{matrix}\right\}$	$\left\{\begin{matrix}3,11,14,15,\\21,22,23\end{matrix}\right\}$	
$\mathbb{T}(P_i)$	$\{a,b\}$	$\{a,c\}$	$\{b,d\}$	

must have occurred after $\max D_1$ — the end of the first phase — and therefore it cannot be an eviction associated with the first phase.

4 Deterministic Marking Algorithms

In a manner similar to [6], based on the phase partitioning of Section 3, we define a class of online algorithms called *marking* algorithms.

Definition 1. *During the request sequence an item* $e \in \bigcup_t M(t)$ *is called* marked *(see Figure 2 for a definition of* $M(t)$*). An online caching algorithm that never evicts marked items is called a* marking algorithm*.*

Remarks:

1. The phase partitioning and dynamic update of the set of marked items can be performed in an online fashion (as given in the algorithm of Fig. 2).
2. At any point in time, the cache can accommodate all marked items.
3. Unlike the marking algorithms of [6], it is not true that immediately after $\max D_i$ all marks of the ith phase are erased. Only the marked items of types $t \in \mathbb{T}(P_i)$ will have their markings erased immediately after $\max D_i$.

For a specific algorithm, at any point in time during the request sequence, a type t that has more than k items in the cache is called *represented in the companion cache*. Note that for marking algorithms, a type is in $\mathbb{T}(P_i)$ if and only if it is represented in the companion cache at $\max D_i$ or it is the type of the item that ended phase i.

Proposition 3. *The number of faults of any marking algorithm on requests whose indices are in* P_i *is at most* $n(k+1)+k = (n+1)(k+1)-1$*.*

Proof. Each item e of type t requested in request index $\ell \in P_i$, is marked and is not evicted until after $\max D_i$. We note that $|\mathbb{T}(P_i)| \le n+1$ since at most n types are represented in the companion cache, and the type of the item whose request ends the phase may also be in $\mathbb{T}(P_i)$. Thus, $|\mathbb{I}(P_i)| \le (n+1)k+n$. □

We conclude from Proposition 3 and Proposition 2:

Theorem 2. *Any marking algorithm is* $(n+1)(k+1)-1$ *competitive.*

Since the marking property can be realized by deterministic algorithms, we conclude that the deterministic competitive ratio of the (n,k)-companion caching problem is $(n+1)(k+1)-1$.

5 Randomized Marking Algorithms

In this section we present an $O(\log n \log k)$ competitive randomized marking algorithm. The building blocks of our randomized algorithms are the following three eviction strategies:

On a fault on an item of type t:

Type Eviction. Evict an item chosen uniformly at random among all unmarked items of type t in the cache.

Cache-Wide Eviction. Let T be the set of types represented in the companion cache, let U be the set of all unmarked items in the cache whose type is in $T \cup \{t\}$. Evict an item chosen uniformly at random from U.

Skewed Cache-Wide Eviction. Let T be the set of types represented in the companion cache, let $T' \subset T \cup \{t\}$ be the set of types with at least one unmarked item in the cache. Choose t' uniformly at random from T', let U be the set of all unmarked items of type t', and evict an item chosen uniformly at random from U.

Remarks:

- Type eviction may not be possible as there may be no unmarked items of type t in the cache.
- Cache-wide eviction and skewed cache-wide eviction are always possible, if there are no unmarked pages of types represented in the companion cache and no unmarked pages of type t in the cache then the fault would have ended the phase.

The algorithms we use are:

Algorithm TP_1. Given a request for item e of type t, not in the cache: Update all phase related status variables (as in the algorithm of Figure 2).

- If t is not represented in the companion cache and there are unmarked items of type t, use type-eviction.
- Otherwise — use cache-wide eviction.

Algorithm TP_2. Given a request for item e of type t, not in the cache: Update all phase related status variables (as in the algorithm of Figure 2). Let the current request index be $j \in D_i$, $i \geq 1$.

- If t is not represented in the companion cache and there are unmarked items of type t, use type-eviction.
- If t is represented in the companion cache, $e \in \mathbb{I}(P_{i-1})$, and there are unmarked items of type t, use type eviction.
- Otherwise — use skewed cache-wide eviction.

Algorithm TP. If $k < n$ use TP_1, otherwise, use TP_2.

Theorem 3. *Algorithm TP is $O(\log n \log k)$ competitive.*

Due to lack of space we only give an overview of the proof in the next section.

5.1 Proof Overview

We give an analogue to the definitions of new and stale pages used in the analysis of the randomized marking paging algorithm of [4].

Definition 2. *For phase i and type t, denote by i^{-t} the largest index $j < i$ such that $t \in \mathbb{T}(P_j)$. If no such j exists we denote $i^{-t} = 0$, and use the convention that $P_0 = \emptyset$. Similarly, i^{+t} is the smallest index $j > i$ such that $t \in \mathbb{T}(P_j)$. If no such index exists, we set $i^{+t} =$ "∞", and use the convention that $P_\infty = \emptyset$.*

Definition 3. *An item e of type t is called* new *in P_i if $e \in \mathbb{I}(P_i) \setminus \mathbb{I}(P_{i^{-t}})$. We denote by $g_{t,i}$ the number of new items of type t in P_i. Note that if $t \notin \mathbb{T}(P_i)$ then $g_{t,i} = 0$.*

Let i_{end} denote the index of the last *completed* phase.

Definition 4. *For $t \in \mathbb{T}(P_i)$, let $L_{t,i} = \mathbb{I}(P_i) \cap \{\text{items of type } t\}$. Note that $|L_{t,i}| \geq k$. Define*

$$\ell_{t,i} = \begin{cases} |L_{t,i}| - k & i < i_{end} \wedge t \in \mathbb{T}(P_i) \setminus \mathbb{T}(P_{i+1}), \\ 0 & \text{otherwise.} \end{cases}$$

We use the above definitions to give an amortized lower bound on the cost to Opt of dealing with the sequence σ:

Lemma 1. *There exist $C > 0$ such that for any request sequence σ,*

$$\text{cost}_{\mathsf{Opt}}(\sigma) \geq C \max\Big\{ \sum_{i \leq i_{\text{end}}} \sum_{t \in P_i} g_{t,i}, \sum_{i \leq i_{\text{end}}} \sum_{t \in P_i} \ell_{t,i} \Big\}. \qquad (1)$$

Proof (sketch). Consider a phase P_j and the set $\mathbb{T}(P_j)$, for every $t \in \mathbb{T}(P_j)$ there exist phases $P_{j^{-t}}$ and $P_{j^{+t}}$, except for some constant number of phases and types.

The number of new items $g_{j^{+t},t}$, $t \in \mathbb{T}(P_j)$, can be associated with a cost to opt, as follows: If an item of type t is not present in the adversary cache at the end of phase P_j, then the adversary must have faulted on this item when it was subsequently requested during phase $P_{j^{+t}}$. If some ℓ of these elements were in the cache at the end of phase P_j (irrespective of their type), then we know that (a) these elements were not requested in P_j — as they are new in $P_{j^{+t}}$ for some t, and (b) under the assumption that opt is lazy — all of these ℓ items have been in opt's cache since (at least) the end of (some) $P_{j^{-t}}$ (where t depends on the specific item in question).

This means that the sequence of requests comprising P_j must have had ℓ items that were in the cache (at least immediately after being requested), yet must have been subsequently evicted before the end of phase P_j.

We also need to avoid overcounting the same evictions multiple times. We argue that we do not do so because these evictions are all accounted for before the end of phase P_j. Now, although in general the requests associated with two phases can be interleaved — requests to items of type $t \in \mathbb{T}(P_j)$ can only occur after the end of phase $P_{j^{-t}}$.

This (almost) proves the 1st lower bound in (1), with the caveat that the proof above seemingly requires an additive constant. A more refined argument shows that this additive constant is not really required.

The proof of the 2nd lower bound in (1) requires similar arguments and is omitted. □

We will give upper bounds (algorithms) that belong to a restricted family of randomized algorithms, specifically *uniform type preference* algorithms defined below. The main advantage of using such algorithms is that their analysis is simplified as they have the property that while dealing with requests σ_j, $j \in D_i$, the companion cache contains only items of types in $\mathbb{T}(P_i) \cup \mathbb{T}(P_{i-1})$.

Definition 5. *A* type preference *algorithm is a marking algorithm such that when a fault occurs on an item of a type that is not represented in the companion cache, it evicts an item of the same type, if this is possible.*

Definition 6. *A* uniform type preference *algorithm is a randomized type preference algorithm maintaining the invariant that at any point in time between request indices* $1 + \max D_{i^{-t}}$ *and* $\max D_i$*, inclusive, and any type* $t \in \mathbb{T}(P_i)$*, all unmarked items of type* t *in* $\mathbb{I}(P_{i^{-t}})$ *are equally likely to be in the cache.*

Note that both TP_1 and TP_2 are uniform type preference algorithms.

We use a charge-based amortized analysis to compute the online cost of dealing with a request sequence σ. We charge the expected cost of all but a constant number of requests in σ to at least one of two "charge counts", charge(D_i) and/or charge(P_j) for some $1 \leq i \leq j \leq i_{\text{end}}$. The total cost associated with the online algorithm is bounded above by a constant times

$\sum_{1\le i\le i_{\text{end}}} \text{charge}(D_i) + \sum_{1\le i\le i_{\text{end}}} \text{charge}(P_i)$, excluding a constant number of requests.

Other than a constant number of requests, every request $\sigma_\ell \in \sigma$ has $\ell \in D_{i_1} \cup P_{i_2}$ for some $1 \le i_1 \le i_2 \le i_{\text{end}}$.

We use the following strategy to charge the cost associated with this request to one (or more) of the charge(D_i), charge(P_j):

1. If $\ell \in P_i$ and type$(\sigma_\ell) \in \mathbb{T}(P_i) \setminus \mathbb{T}(P_{i-1})$ then we charge the (expected) cost of σ_ℓ to charge(P_i). These charges can be amortized against the cost of Opt to deal with σ_ℓ. This amortization is summarized in Proposition 5 (for any uniform type preference algorithm).
2. If $\ell \in D_i$ and type$(\sigma_\ell) \in \mathbb{T}(P_{i-1})$ then we charge the (expected) cost of σ_ℓ to charge(D_i). These charges will be amortized against the cost of Opt to within a poly-logarithmic factor.

To compute the expected cost of a request σ_ℓ, $\ell \in D_i$, type$(\sigma_\ell) \in \mathbb{T}(P_{i-1})$, we introduce an analogue to the concept of "holes" used in [4]. In [4] holes were defined to be stale pages that were evicted from the cache.

Definition 7. *We define the number of holes during D_i, h_i, to be the maximum over the indices $j \in D_i$ of the total number of items of types in $\mathbb{T}(P_{i-1})$ that were requested in $P_{t,i-1}$ but are not cached when the jth request is issued.*

Proposition 4. *Consider a marking algorithm, a phase i, a type $t \in \mathbb{T}(P_i) \setminus \mathbb{T}(P_{i-1})$, and a request index* $\max D_{i^{-t}} < j \le \max D_i$. *Let H be the set of items of type t that were requested in $P_{i^{-t}}$ and evicted afterward without being requested again, up to request index j (inclusive). Then $|H| \le \hat{g}_{t,i} + \ell_{t,i^{-t}}$, where $\hat{g}_{t,i} \le g_{t,i}$ is the number of new items of type t requested after* $\max D_{i^{-t}}$ *and up to time j (inclusive).* □

Proposition 5. *For a uniform type preference algorithm, the expected number of faults on request indices in P_i for items of type $t \in \mathbb{T}(P_i) \setminus \mathbb{T}(P_{i-1})$ is at most* $(1 + H_{n+k})(g_{t,i} + \ell_{t,i^{-t}})$. I.e., *charge*$(P_i) \le (1 + H_{n+k})(g_{t,i} + \ell_{t,i^{-t}})$.

Proof. Fix a type $t \in \mathbb{T}(P_i) \setminus \mathbb{T}(P_{i-1})$. There are $g_{t,i}$ faults on new items of type t, the rest of the faults are on items in $L_{t,i^{-t}}$ that were evicted before being requested again. By Proposition 4, the number of items in $L_{t,i^{-t}}$ that are not in the cache at any point of time is at most $\hat{g}_{t,i} + \ell_{t,i^{-t}} \le g_{t,i} + \ell_{t,i^{-t}}$. For any a, b in $L_{t,i^{-t}}$ that have not been requested after $\max D_{i^{-t}}$, the probability that a has been evicted since $1 + \max D_{i^{-t}}$ is equal to the probability that b has been evicted since $1 + \max D_{i^{-t}}$.

Let r denote the number of items in $L_{t,i^{-t}}$ that have been requested since after $\max D_{i^{-t}}$. There are $|L_{t,i^{-t}}| - r$ unmarked items of $L_{t,i^{-t}}$. The probability that an item of $L_{t,i^{-t}}$ is cached is therefore $(g_{t,i} + \ell_{t,i^{-t}})/(|L_{t,i^{-t}}| - r)$. Thus, the expected number of faults on requests indices in P_i for items in $L_{t,i^{-t}}$ is at most

$$\sum_{r=0}^{|L_{t,i^{-t}}|-1} \frac{g_{t,i} + \ell_{t,i^{-t}}}{|L_{t,i^{-t}}| - r} \le (g_{t,i} + \ell_{t,i^{-t}}) H_{|L_{t,i^{-t}}|} \le (g_{t,i} + \ell_{t,i^{-t}}) H_{n+k}. \qquad \square$$

Proposition 6. *A type preference algorithm has the following properties:*

1. *During D_i, only types in $\mathbb{T}(P_{i-1}) \cup \mathbb{T}(P_i)$ may be represented in the companion cache.*
2. *During D_i, when a type $t \in \mathbb{T}(P_i) \setminus \mathbb{T}(P_{i-1})$ becomes represented in the companion cache, there are no unmarked cached items of type t, and t stays represented in the companion cache until* $\max D_i$*, inclusive.* □

Proposition 7. *For a type preference algorithm,*

$$h_i \leq \sum_{t \in \mathbb{T}(P_i) \setminus \mathbb{T}(P_{i-1})} (g_{t,i} + \ell_{t,i^{-t}}) + \sum_{t \in \mathbb{T}(P_{i-1})} g_{t,(i-1)^{+t}}. \qquad (2)$$

Definition 8. *At any point during D_i, call a type $t \in \mathbb{T}(P_{i-1})$ that has unmarked items in the cache and is represented in the companion cache an* active type*. Call an unmarked item $e \in \mathbb{I}(P_{i-1})$ of an active type an* active item*.*

Proposition 8. *The following properties hold for type preference algorithms:*

1. *During D_i, the set of active types is monotone decreasing w.r.t. containment.*
2. *During D_i, the set of active items is monotone decreasing w.r.t. containment.*

Proposition 9. *For TP_1, charge(D_i) — The expected number of faults on request indices in D_i to types in $\mathbb{T}(P_{i-1})$ — is at most $h_i(1 + H_{k+1}(1 + H_{(n+1)(k+1)}))$.*

Proof. First, we count the expected number of faults on items in $\cup_{t \in \mathbb{T}(P_{i-1})} L_{t,i-1}$. By Proposition 8, the set of active items is monotone decreasing, where an item becomes inactive either by being marked, or because its type is no longer represented in the companion cache. Let $\langle m_j \rangle_{j=1,\ldots,w}$, be a sequence, indexed by the *event* index, of the number of active items. An event is either when an active item is requested, or when an active type t becomes inactive by being no longer represented in the companion cache (it is possible that one request generates two events, one from each case).

If the jth event is a request for active item, then $m_{j+1} = m_j - 1$. Otherwise, if the jth event is the event of type t becoming inactive, and before that event there were b active items of type t, then $m_{j+1} = m_j - b$.

In the first case, the expected cost of the request, conditioned on m_j, is at most h_i/m_j. In the second case, there are b items of type t that became inactive, each had probability of $\frac{h_i}{m_j}$ of not being in the cache at that moment. This means that the expected number of items among the up-until now active items of type t, that are not in the cache, at this point in time, is $\frac{h_i b}{m_j}$.

Let g_t denote the number of new items of $P_{(i-1)^{+t}}$ (Def. 3) requested during D_i ($g_t \leq g_{t,(i-1)^{+t}}$). After type t becomes inactive, the number of items among $L_{t,i-1}$ that are not in the cache can increase only when a new item of type t is requested. Therefore the expected number of items among $L_{t,i-1}$ that are not in the cache, after the jth event (the event when t became inactive), is at most $\frac{h_i b}{m_j} + g_t$.

Because of the uniform type eviction property of TP_1, the probability that an item in $L_{t,i-1}$ is not in the cache is the expected number of items among $L_{t,i-1}$, and not in the cache, divided by the number of unmarked items among $L_{t,i-1}$, and therefore the expected number of faults on items of $L_{t,i-1}$ after the jth event is at most

$$\sum_{a=1}^{b}(\frac{h_i b}{m_j}+g_t)\cdot\frac{1}{a}=(\frac{h_i b}{m_j}+g_t)H_b.$$

Note that $b \le k+1$, and $\sum_{t\in P_{i-1}} g_t \le h_i$, and so the expected number of faults on items $e \in \cup_{t\in\mathbb{T}(P_{i-1})} L_{t,i-1}$, conditioned on the sequence $\langle m_j\rangle_j$ is at most

$$h_i H_{k+1} + h_i \sum_j \frac{(m_j - m_{j-1})H_{k+1}}{m_j} \tag{3}$$

The sequence $\langle m_j\rangle_j$ is itself a random variable, but we can give an upper bound on the expected number of faults on items $e \in \cup_{t\in\mathbb{T}(P_{i-1})} L_{t,i-1}$ by bounding the *maximum* of (3) over all feasible sequences $\langle m_j\rangle_j$. The worst case for (3) will be when $\langle m_j\rangle_j = \langle (n+1)(k+1)-j\rangle_{j=1}^{(n+1)(k+1)-1}$. Thus,

$$h_i H_{k+1}(1+\sum_j \frac{(m_j - m_{j+1})}{m_j}) \le h_i H_{k+1}(1+H_{(n+1)(k+1)})$$

We are left to add faults on new items of types in $\mathbb{T}(P_{i-1})$. There are at most $\sum_{t\in\mathbb{T}(P_{i-1})} g_{t,(i-1)^{+t}} \le h_i$ such faults. □

Lemma 2. TP_1 *is* $O(\log k \max\{\log n, \log k\})$ *competitive.*

Proof. Each fault is counted by either charge(P_i) (Proposition 5) or charge(D_i) (Proposition 9) (faults on request indices in D_i for items of type in $\mathbb{T}(P_{i-1})\setminus\mathbb{T}(P_i)$ are counted twice), and by Lemma 1, we have that the expected number of faults of TP_1 is at most $O(H_{k+1}H_{(n+1)(k+1)})\,\mathrm{cost}_{\mathsf{Opt}}$. □

For algorithm TP_2, we have the following result:

Lemma 3. TP_2 *is* $O(\log n \max\{\log n, \log k\})$ *competitive.* □

The proof, which uses ideas similar to the proof of Lemma 2, has been omitted.

Proof (of Theorem 3). Follows immediately from Lemma 2 and Lemma 3. □

Unfortunately, the competitive ratio of type preference algorithms is always $\Omega(\log n \log k)$. The proof of this claim is omitted in this version for lack of space.

References

[1] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234:203–218, 2000. 500

[2] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966. 500

[3] M. Brehob, R. Enbody, E. Torng, and S. Wagner. On-line restricted caching. In *Proceedings of the 12th Symposium on Discrete Algorithms*, pages 374–383, 2001. 499, 500, 501

[4] A. Fiat, R. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991. 500, 501, 502, 506, 508

[5] N. Jouppi. Improving direct-mapped cache by the addition of small fully-associative cache and prefetch buffer. *Proc. of the 17th International Symposiuom on Computer Architecture*, 18(2):364–373, 1990.

[6] A. Karlin, M. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988. 501, 502, 504

[7] L. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *J. Algorithms*, 6:816–825, 1991. 500

[8] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communication of the ACM*, 28:202–208, 1985. 500

[9] A. Seznec. A case for two-way skewed-associative caches. In *Proc. of the 20th International Symposuim on Computer Architecture* pages 169–178, 1993.